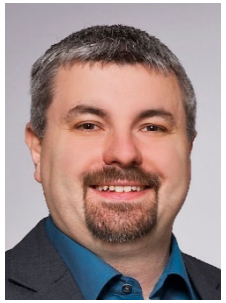
Programming OpenMP

Christian Terboven

Michael Klemm



Agenda (in total 7 Sessions)

- **Session 1: OpenMP Introduction**
  - → Welcome
  - → OpenMP Overview
  - → Parallel Region
  - → Worksharing
  - → Scoping
  - → Tasking (short introduction)
  - → Executing OpenMP programs
  - → Homework assignments ☺
  - → Compile and run on Perlmutter CPUs

- Session 2: Tasking
- Session 3: Optimization for NUMA and SIMD
- Session 4: What Could Possibly Go Wrong Using OpenMP
- Session 5: Introduction to Offloading with OpenMP
- Session 6: Advanced OpenMP Offloading Topics
- Session 7: Selected / Remaining Topics

Programming OpenMP

An Overview Of OpenMP

Christian Terboven

Michael Klemm

History

- De-facto standard for Shared-Memory Parallelization.

- 1997: OpenMP 1.0 for FORTRAN
- 1998: OpenMP 1.0 for C and C++
- 1999: OpenMP 1.1 for FORTRAN
- 2000: OpenMP 2.0 for FORTRAN
- 2002: OpenMP 2.0 for C and C++
- 2005: OpenMP 2.5 now includes both programming languages.

- 05/2008: OpenMP 3.0
- 07/2011: OpenMP 3.1

- 07/2013: OpenMP 4.0
- 11/2015: OpenMP 4.5

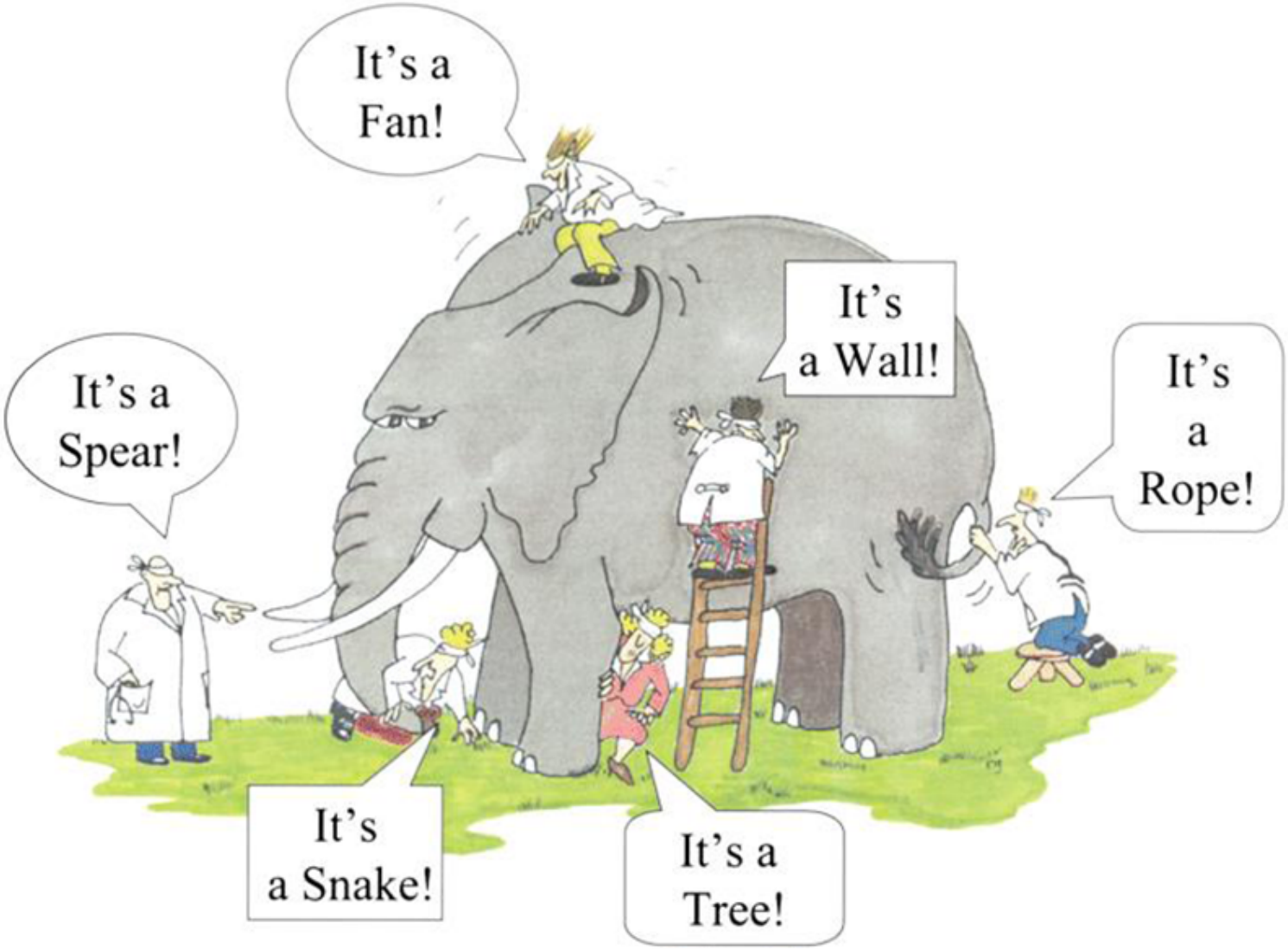
- 11/2018: OpenMP 5.0
- 11/2020: OpenMP 5.1
- 11/2021: OpenMP 5.2

http://www.OpenMP.org

RWTH Aachen University is a member of the OpenMP Architecture Review Board (ARB) since 2006.
Main topics:
- Affinity
- Tasking
- Tool support
- Accelerator support

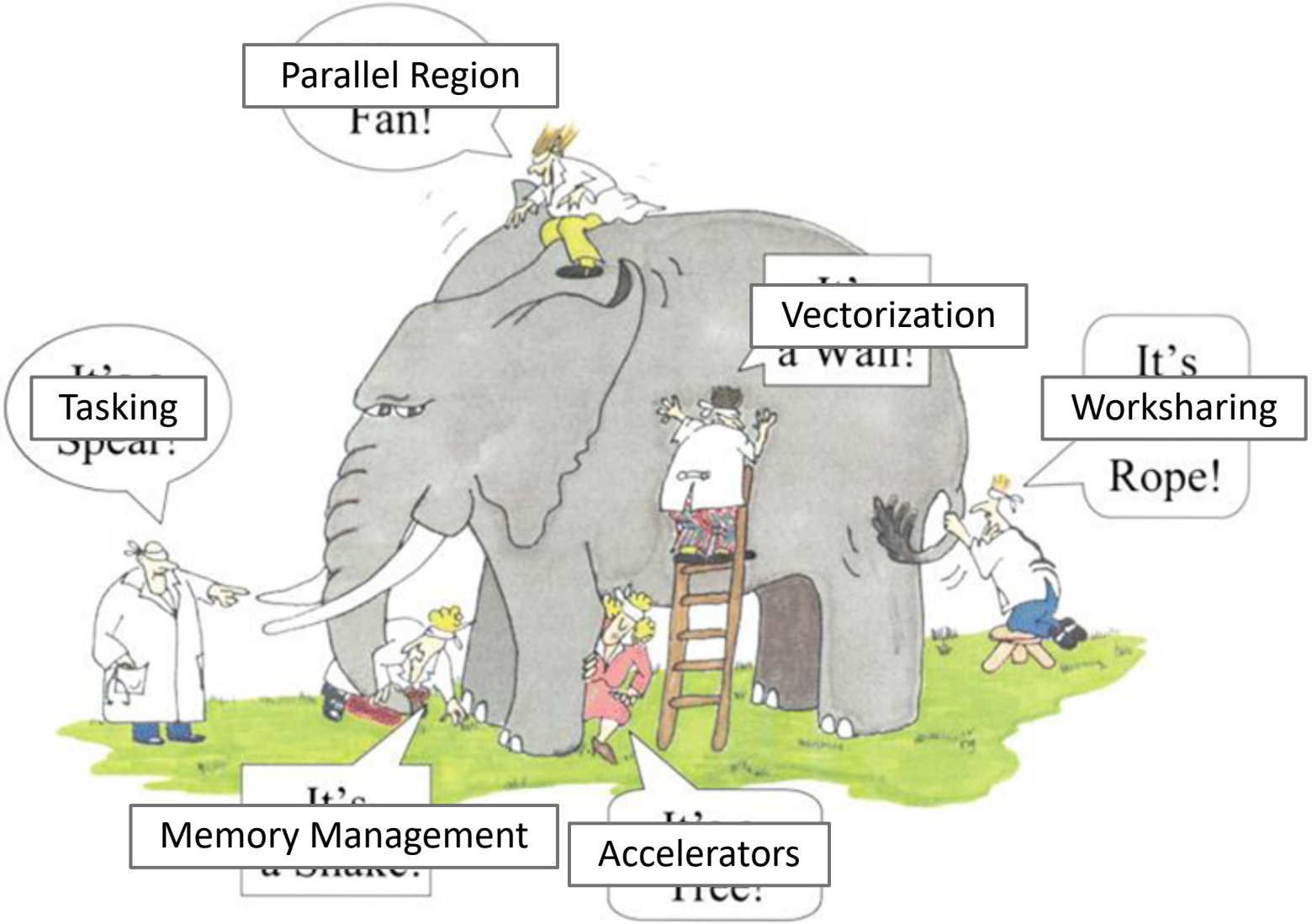
What is OpenMP?

- Parallel Region & Worksharing
- Tasking
- SIMD / Vectorization
- Accelerator Programming
- Memory Management
- …

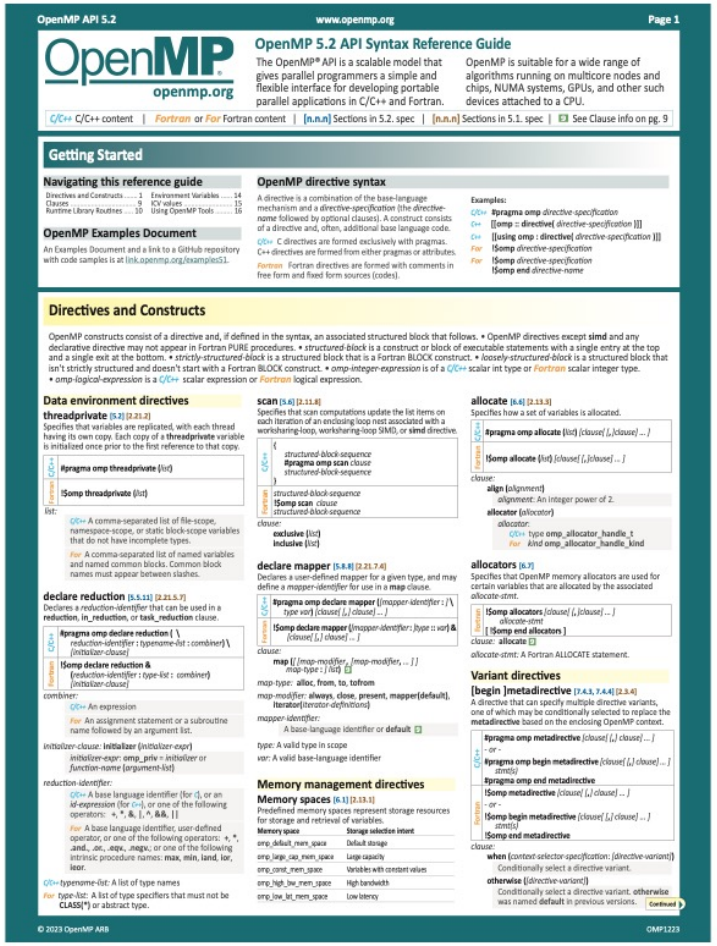


What is OpenMP?

- Parallel Region & Worksharing
- Tasking
- SIMD / Vectorization
- Accelerator Programming
- Memory Management
- …



Get your C/C++ and Fortran Reference Guide!
Covers all of OpenMP 5.2!



OpenMP API 5.2 www.openmp.org Page 1

OpenMP 5.2 API Syntax Reference Guide

The OpenMP® API is a scalable model that gives parallel programmers a simple and flexible interface for developing portable parallel applications in C/C++ and Fortran.

OpenMP is suitable for a wide range of algorithms running on multicore nodes and chips, NUMA systems, GPUs, and other such devices attached to a CPU.

C/C++ C/C++ content | Fortran or For Fortran content | [n.n.n] Sections in 5.2. spec | [n.n.n] Sections in 5.1. spec | See Clause info on pg. 9

Getting Started

Navigating this reference guide

Directives and Constructs 1
Clauses 9
Runtime Library Routines 10
Environment Variables 14
ICV values 15
Using OpenMP Tools 16

OpenMP Examples Document

An Examples Document and a link to a GitHub repository with code samples is at link.openmp.org/examples51.

OpenMP directive syntax

A directive is a combination of the base-language mechanism and a *directive-specification* (the *directive-name* followed by optional clauses). A construct consists of a directive and, often, additional base language code.

C/C++ C directives are formed exclusively with pragmas. C++ directives are formed from either pragmas or attributes.

Fortran Fortran directives are formed with comments in free form and fixed form sources (codes).

Examples:

C/C++ **#pragma omp** *directive-specification*
C++ **[[omp :: directive(** *directive-specification* **)]]**
C++ **[[using omp : directive(** *directive-specification* **)]]**
For **!$omp** *directive-specification*
For **!$omp** *directive-specification*
!$omp end *directive-name*

Directives and Constructs

OpenMP constructs consist of a directive and, if defined in the syntax, an associated structured block that follows. • OpenMP directives except **simd** and any declarative directive may not appear in Fortran PURE procedures. • *structured-block* is a construct or block of executable statements with a single entry at the top and a single exit at the bottom. • *strictly-structured-block* is a structured block that is a Fortran BLOCK construct. • *loosely-structured-block* is a structured block that isn't strictly structured and doesn't start with a Fortran BLOCK construct. • *omp-integer-expression* is of a C/C++ scalar int type or Fortran scalar integer type. • *omp-logical-expression* is a C/C++ scalar expression or Fortran logical expression.

Data environment directives

threadprivate [5.2] [2.21.2]
Specifies that variables are replicated, with each thread having its own copy. Each copy of a **threadprivate** variable is initialized once prior to the first reference to that copy.

C/C++ **#pragma omp threadprivate** (*list*)
Fortran **!$omp threadprivate** (*list*)

list:
C/C++ A comma-separated list of file-scope, namespace-scope, or static block-scope variables that do not have incomplete types.
For A comma-separated list of named variables and named common blocks. Common block names must appear between slashes.

declare reduction [5.5.11] [2.21.5.7]
Declares a *reduction-identifier* that can be used in a **reduction**, **in_reduction**, or **task_reduction** clause.

C/C++ **#pragma omp declare reduction** (\
reduction-identifier : *typename-list* : *combiner*) \
{initializer-clause}
Fortran **!$omp declare reduction** &
{*reduction-identifier* : *type-list* : *combiner*}
{initializer-clause}

combiner:
C/C++ An expression
For An assignment statement or a subroutine name followed by an argument list.

initializer-clause: **initializer** (*initializer-expr*)
initializer-expr: **omp_priv** = *initializer* or *function-name* (*argument-list*)

reduction-identifier:
C/C++ A base language identifier (for *id*), or an *id-expression* (for C++), or one of the following operators: +, -, *, &, |, ^, &&, ||
For A base language identifier, user-defined operator, or one of the following operators: +, *, .and., .or., .eqv., .neqv.; or one of the following intrinsic procedure names: **max**, **min**, **iand**, **ior**, **ieor**.

C/C++ *typename-list*: A list of type names
For *type-list*: A list of type specifiers that must not be **CLASS(*)** or abstract type.

scan [5.6] [2.11.8]
Specifies that scan computations update the list items on each iteration of an enclosing loop nest associated with a worksharing-loop, worksharing-loop SIMD, or **simd** directive.

C/C++ {
structured-block-sequence
#pragma omp scan *clause*
structured-block-sequence
}
Fortran *structured-block-sequence*
!$omp scan *clause*
structured-block-sequence

clause:
exclusive (*list*)
inclusive (*list*)

declare mapper [5.8.8] [2.21.7.4]
Declares a user-defined mapper for a given type, and may define a *mapper-identifier* for use in a **map** clause.

C/C++ **#pragma omp declare mapper** (*mapper-identifier* : \
type var) *{clause[[,] clause] ... }*
Fortran **!$omp declare mapper** (*mapper-identifier* : *type* :: *var*) &
{clause[[,] clause] ... }

clause:
map ([*map-modifier*, *map-modifier*, ...]]
map-type :] *list*)

map-type: **alloc**, **from**, **to**, **tofrom**
map-modifier: **always**, **close**, **present**, **mapper(default)**, **iterator(***iterator-definitions***)**
mapper-identifier:
A base-language identifier or **default**

type: A valid type in scope
var: A valid base-language identifier

Memory management directives

Memory spaces [6.1] [2.13.1]
Predefined memory spaces represent storage resources for storage and retrieval of variables.

Memory space	Storage selection intent
omp_default_mem_space	Default storage
omp_large_cap_mem_space	Large capacity
omp_const_mem_space	Variables with constant values
omp_high_bw_mem_space	High bandwidth
omp_low_lat_mem_space	Low latency

allocate [6.6] [2.13.3]
Specifies how a set of variables is allocated.

C/C++ **#pragma omp allocate** (*list*) *{clause[[,] clause] ... }*
Fortran **!$omp allocate** (*list*) *{clause[[,] clause] ... }*

clause:
align (*alignment*)
alignment: An integer power of 2.
allocator (*allocator*)
allocator:
C/C++ type **omp_allocator_handle_t**
For *kind* **omp_allocator_handle_kind**

allocators [6.7]
Specifies that OpenMP memory allocators are used for certain variables that are allocated by the associated *allocate-stmt*.

Fortran **!$omp allocators** *{clause} [,] clause] ... }*
allocate-stmt
[**!$omp end allocators**]

clause: **allocate**

allocate-stmt: A Fortran ALLOCATE statement.

Variant directives

[begin]metadirective [7.4.3, 7.4.4] [2.3.4]
A directive that can specify multiple directive variants, one of which may be conditionally selected to replace the **metadirective** based on the enclosing OpenMP context.

C/C++ **#pragma omp metadirective** *{clause[[,] clause] ... }*
- or -
#pragma omp begin metadirective *{clause[[,] clause] ... }*
stmt(s)
#pragma omp end metadirective
Fortran **!$omp metadirective** *{clause[[,] clause] ... }*
- or -
!$omp begin metadirective *{clause[[,] clause] ... }*
stmt(s)
!$omp end metadirective

clause:
when (*context-selector-specification*: {*directive-variant*})
Conditionally select a directive variant.
otherwise ({*directive-variant*})
Conditionally select a directive variant. **otherwise** was named **default** in previous versions.

Continued

© 2023 OpenMP ARB OMP1223

OpenMP API 5.2 C/C++ C/C++ content | Fortran or For Fortran content | [n.n.n] Sections in 5.2. spec | [n.n.n] Sections in 5.1. spec | See Clause info on pg. 9 2

Directives and Constructs (continued)

[begin]declare variant [7.5.4-5] [2.3.5]
Declares a specialized variant of a base function and the context in which it is used.

C/C++ **#pragma omp declare variant**(*variant-func-id*) \
clause [[[,] clause] ...]
|**#pragma omp declare variant**(*variant-func-id*) \
clause [[[,] clause] ...]
[...]]
function definition or declaration
- or -
#pragma omp begin declare variant *clause-match*
declaration-definition-seq
#pragma omp end declare variant
Fortran **!$omp declare variant** (*base-proc-name* :) &
variant-proc-name) *clause [[[,] clause] ...]*

clause:
adjust_args (*adjust-op* : *argument-list*)
adjust-op: **nothing**, **need_device_ptr**
append_args (*append-op*[[, *append-op* [...]])
append-op: **interop** (
interop-type[ ] , *interop-type* [...]])
enter
link
match (*context-selector-specification*)
REQUIRED. Specifies how to adjust the arguments of the base function when a specified variant function is selected for replacement.

C/C++ *variant-func-id*
The name of a function variant that is a base language identifier, or for C++, a *template-id*.

For *variant-proc-name*
The name of a function variant that is a base language identifier.

clause-match:
match (*context-selector-specification*)
REQUIRED **match** clause

dispatch [7.6] [2.3.6]
Controls whether variant substitution occurs for a function call in the structured block.

C/C++ **#pragma omp dispatch** *{clause [[,] clause] ... }*
function-dispatch-structured-block
Fortran **!$omp dispatch** *{clause [[,] clause] ... }*
function-dispatch-structured-block
[**!$omp end dispatch**]

clause:
depend ({*depend-modifier*, } *dependence-type* : *locator-list*)
device (*omp-integer-expression*)
Identifies the target device that is associated with a device construct.
is_device_ptr (*list*)
list: device pointers
nocontext (*omp-logical-expression*)
If *omp-logical-expression* evaluates to true, the construct is not added to the construct set of the OpenMP context.
novariants (*omp-logical-expression*)
If *omp-logical-expression* evaluates to true, no function variant is selected for the call in the applicable **dispatch** region.
nowait

declare simd [7.7] [2.11.5.3]
Applied to a function or subroutine to enable creation of one or more versions to process multiple arguments using SIMD instructions from a single invocation in a SIMD loop.

C/C++ **#pragma omp declare simd** *{clause[[,] clause] ... }*
|**#pragma omp declare simd** *{clause[[,] clause] ... } [...]*
function definition or declaration
Fortran **!$omp declare simd** (*proc-name*)[*{clause[[,] clause] ... }*]

clause:
aligned (*argument-list* : *alignment*)
Declares one or more list items to be aligned to the specified number of bytes.
alignment: Optional constant positive integer expression
inbranch
linear (*linear-list* : *linear-step*)
notinbranch
simdlen (*length*)
Specifies the preferred number of iterations to be executed concurrently.
uniform (*argument-list*)
Declares arguments to have an invariant value for all concurrent invocations of the function in the execution of a single SIMD loop.

[begin]declare target [7.8.1-7] [2.14.7]
A declarative directive that specifies that variables, functions, and subroutines are mapped to a device.

C/C++ **#pragma omp declare target** (*extended-list*)
- or -
#pragma omp declare target *clause[[,] clause ...]*
- or -
#pragma omp begin declare target \
[clause[[,] clause] ...]
declarations-definition-seq
#pragma omp end declare target
Fortran **!$omp declare target** (*extended-list*)
- or -
!$omp declare target *[clause[[,] clause] ...]*

clause:
device_type (**host** | **nohost** | **any**)
enter (*extended-list*)
A comma-separated list of named variables, procedure names, and named common blocks.
indirect([*invoked-by-fptr*])
Determines if the procedures in an **enter** clause may be invoked indirectly.
link (*list*)
Supports compilation of functions called in a **target** region that refer to the *list* items.

- For the second C/C++ form of **declare target**, at least one clause must be **enter** or **link**.
- For **begin declare target**, the **enter** and **link** clauses are not permitted.

Informational and utility directives

requires [9.2] [2.5.1]
Specifies the features that an implementation must provide in order for the code to compile and to execute correctly.

C/C++ **#pragma omp requires** *clause [[[,] clause] ...]*
Fortran **!$omp requires** *clause [[[,] clause] ...]*

clause:
atomic_default_mem_order (**seq_cst** | **acq_rel** | **relaxed**)
dynamic_allocators
Enables memory allocators to be used in a **target** region without specifying the **uses_allocators** clause on the corresponding **target** construct. (See **target** on page 5 of this guide.)
reverse_offload
Requires an implementation to guarantee that if a **target** construct specifies a **device** clause in which the **ancestor** modifier appears, the **target** region can execute on the parent device of an enclosing **target** region. (See **target** on page 5.)
unified_address
Requires that all devices accessible through OpenMP API routines and directives use a unified address space.
unified_shared_memory
Guarantees that in addition to the requirement of **unified_address**, storage locations in memory are accessible to threads on all available devices.

assume, [begin]assumes [8.3.2-4] [2.5.2]
Provides invariants to the implementation that may be used for optimization purposes.

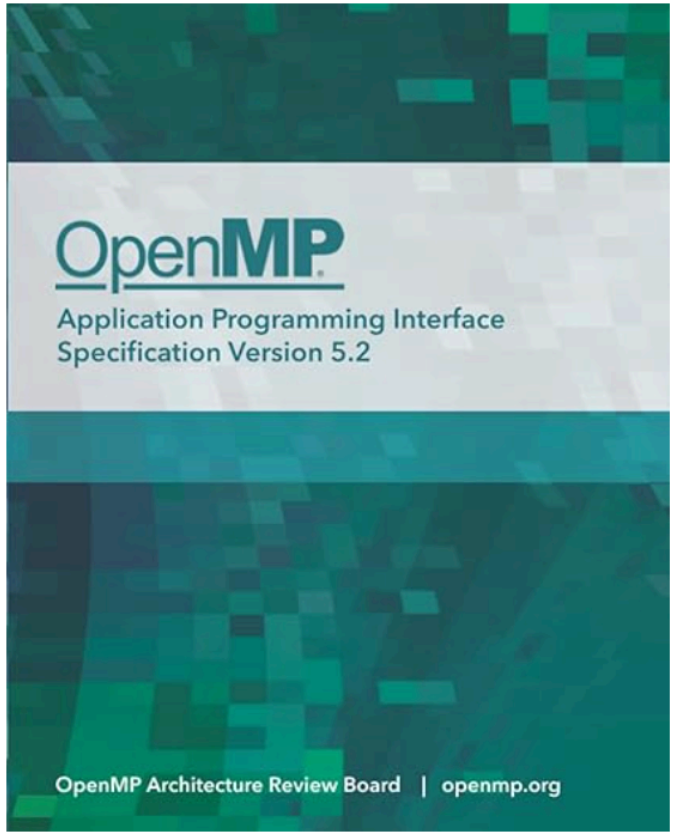
C/C++ **#pragma omp assumes** *clause [[[,] clause] ...]*
- or -
#pragma omp begin assumes *clause [[[,] clause] ...]*
declaration-definition-seq
#pragma omp end assumes
- or -
#pragma omp assume *clause [[[,] clause] ...]*
structured-block
Fortran **!$omp assumes** *clause [[[,] clause] ...]*
- or -
!$omp assume *clause [[[,] clause] ...]*
loosely-structured-block
!$omp end assume
- or -
!$omp assume *clause [[[,] clause] ...]*
strictly-structured-block
[**!$omp end assume**]

clause:
absent (*directive-name* [[, *directive-name*] ...])
Lists directives absent in the scope.
contains (*directive-name* [[, *directive-name*] ...])
Lists directives likely to be in the scope.
holds (*omp-logical-expression*)
An *expression* guaranteed to be true in the scope.
no_openmp
Indicates that no OpenMP code is in the scope.
no_openmp_routines
Indicates that no OpenMP runtime library calls are executed in the scope.
no_parallelism
Indicates that no OpenMP tasks or SIMD constructs will be executed in the scope.

Continued

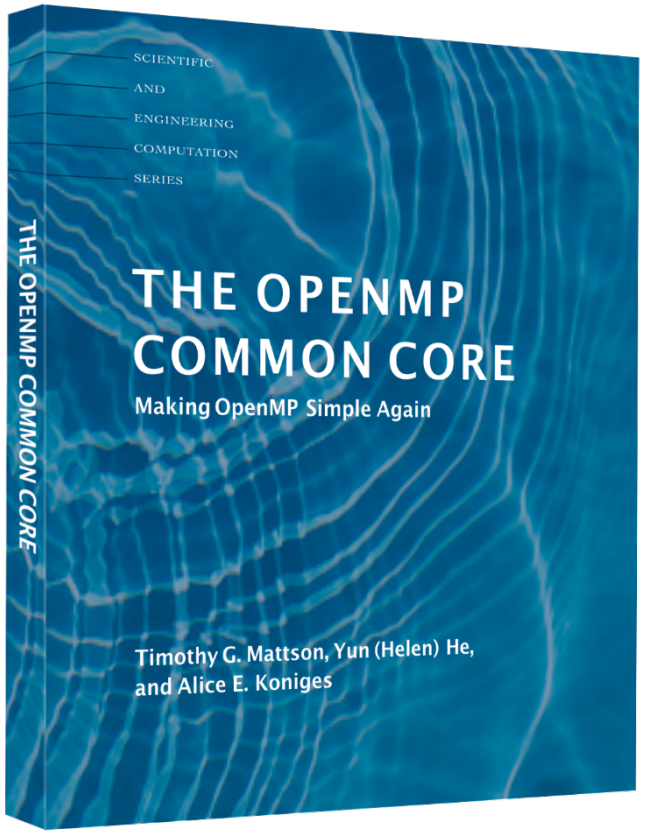
© 2023 OpenMP ARB www.openmp.org OMP1223

Recent Books About OpenMP



A printed copy of the 5.2 specifications, 2021

A book that covers all of the OpenMP 4.5 features, 2017



A book about the OpenMP Common Core, 2019

Programming OpenMP

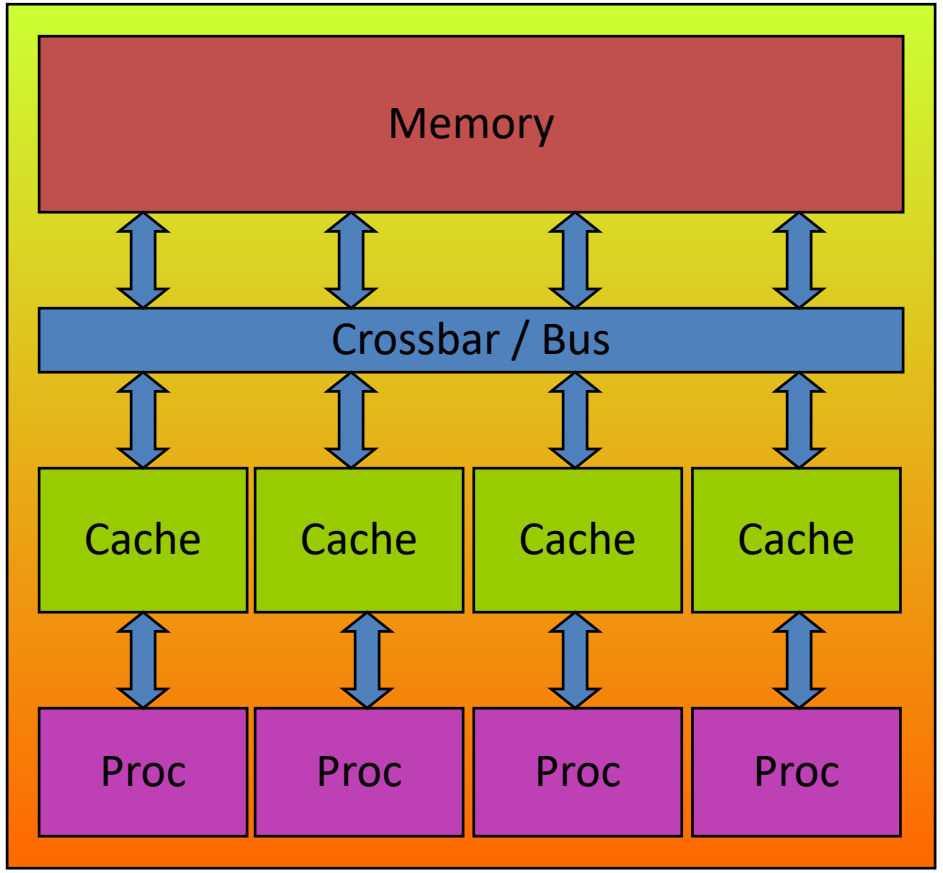
Parallel Region

Christian Terboven

Michael Klemm

OpenMP's machine model

- OpenMP: Shared-Memory Parallel Programming Model.



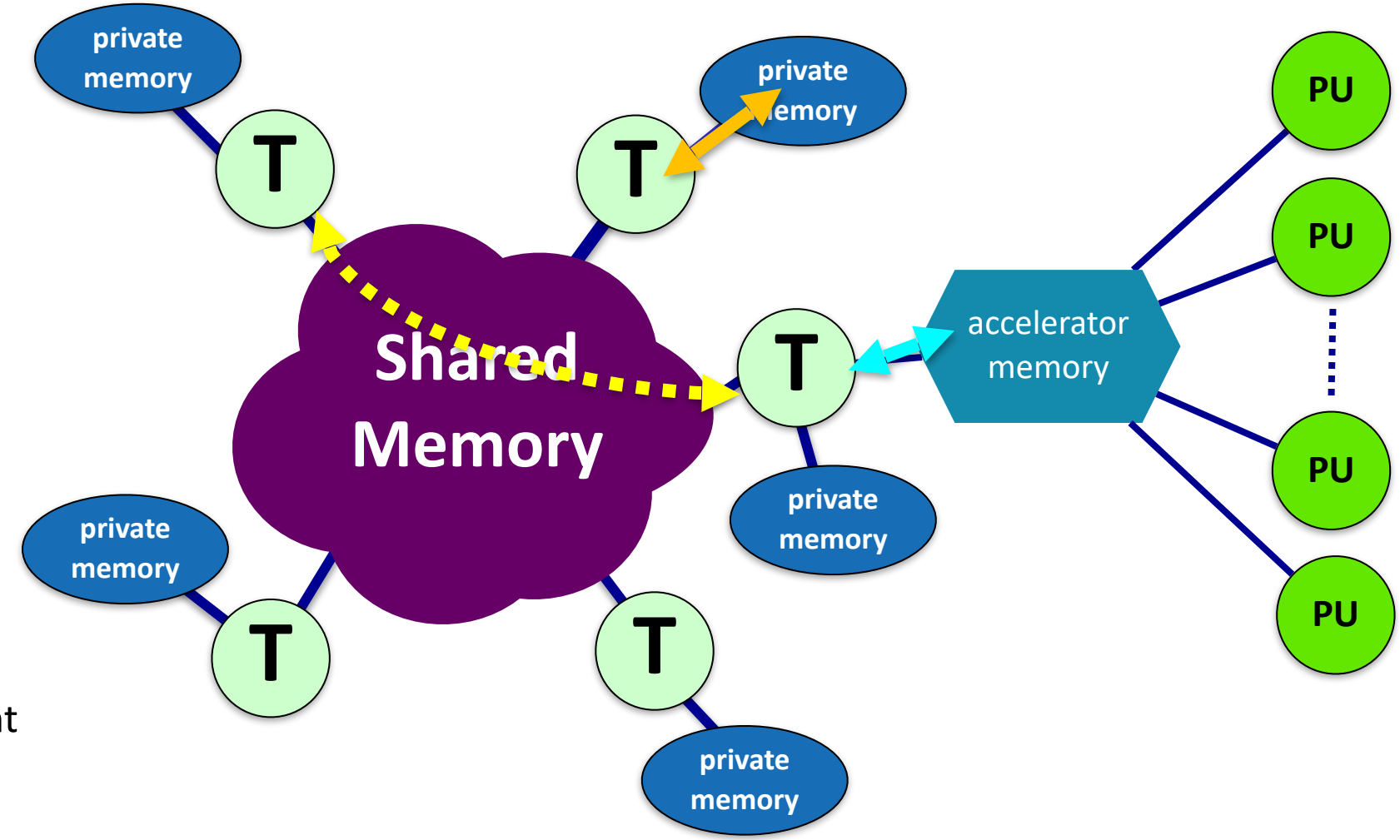
All processors/cores access a shared main memory.

Real architectures are more complex, as we will see later / as we

Parallelization in OpenMP employs multiple threads.

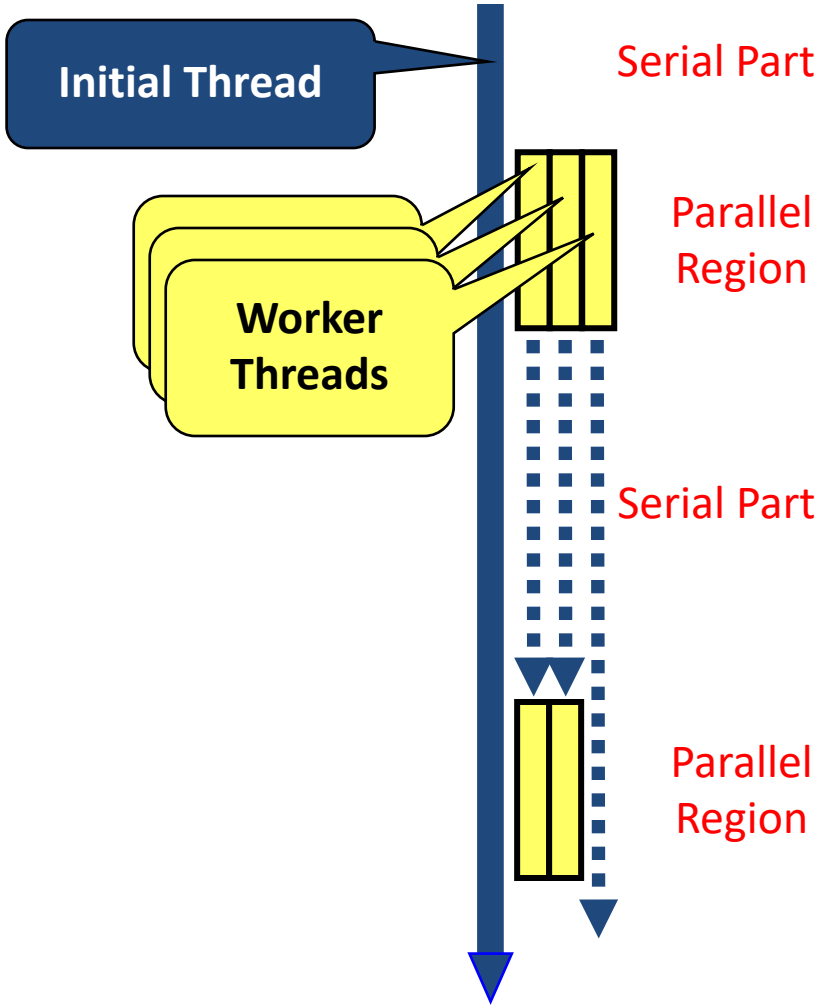
The OpenMP Memory Model

- All threads have access to the same, globally shared memory
- Data in private memory is only accessible by the thread owning this memory
- No other thread sees the change(s) in private memory
- Data transfer is through shared memory and is 100% transparent to the application



The OpenMP Execution Model

- OpenMP programs start with just one thread: The *Initial Thread*.
- *Worker* threads are spawned at *Parallel Regions*, together with the initial thread they form the *Team* of threads.
- In between Parallel Regions the Worker threads are put to sleep. The OpenMP *Runtime* takes care of all thread management work.
- Concept: *Fork-Join*.
- Allows for an incremental parallelization!



Parallel Region and Structured Blocks

- The parallelism has to be expressed explicitly.

C/C++

```
#pragma omp parallel
{
    ...
    structured block
    ...
}
```

Fortran

```
!$omp parallel
    ...
    structured block
    ...
!$omp end parallel
```

- *Structured Block*
  - Exactly one entry point at the top
  - Exactly one exit point at the bottom
  - Branching in or out is not allowed
  - Terminating the program is allowed (abort / exit)

- *Specification of number of threads:*
  - Environment variable: `OMP_NUM_THREADS=…`
  - Or: Via `num_threads` clause: add `num_threads(num)` to the parallel construct

Programming OpenMP

Using OpenMP Compilers

Christian Terboven

Michael Klemm

RWTH AACHEN UNIVERSITY

OpenMP

Production Compilers w/ OpenMP Support

- GCC
- clang/LLVM
- HPE CPE
- AOCC, AOMP, ROCmCC
- Intel Classic and Next-gen Compilers
- IBM XL
- … and many more

- See https://www.openmp.org/resources/openmp-compilers-tools/ for a list

Compiling OpenMP

- Enable OpenMP via the compiler's command-line switches
  - → GCC: `-fopenmp`
  - → clang: `-fopenmp`
  - → HPE/Cray CPE: `-homp` or `-fopenmp`
  - → AOCC, AOCL, ROCmCC: `-fopenmp`
  - → Intel: `-fopenmp` or `–qopenmp` (classic) or `–fiopenmp` (next-gen)
  - → IBM XL: -qsmp=omp
- Switches have to be passed to both compiler and linker:

```
$ gcc [...] -fopenmp -o matmul.o -c matmul.c
$ gcc [...] -fopenmp -o matmul matmul.o
$./matmul 1024
Sum of matrix (serial):   134217728.000000, wall time 0.413975, speed-up 1.00
Sum of matrix (parallel): 134217728.000000, wall time 0.092162, speed-up 4.49
```

Starting OpenMP Programs on Linux

- From within a shell, global setting of the number of threads:

```
export OMP_NUM_THREADS=4
./program
```

- From within a shell, one-time setting of the number of threads:

```
OMP_NUM_THREADS=4    ./program
```

Hello OpenMP World

Programming OpenMP

Worksharing

Christian Terboven

Michael Klemm

RWTH AACHEN UNIVERSITY

For Worksharing

- If only the *parallel* construct is used, each thread executes the Structured Block.
- Program Speedup: *Worksharing*
- OpenMP's most common Worksharing construct: *for*

C/C++

```
int i;
#pragma omp for
for (i = 0; i < 100; i++)
{
   a[i] = b[i] + c[i];
}
```

Fortran

```
INTEGER :: i
!$omp do
DO i = 0, 99
   a[i] = b[i] + c[i]
END DO
```

  - Distribution of loop iterations over all threads in a Team.
  - Scheduling of the distribution can be influenced.

- Loops often account for most of a program's runtime!

Worksharing illustrated

Pseudo-Code
Here: 4 Threads

Serial

```
do i = 0, 99
    a(i) = b(i) + c(i)
end do
```

Thread 1

```
do i = 0, 24
    a(i) = b(i) + c(i)
end do
```

Thread 2

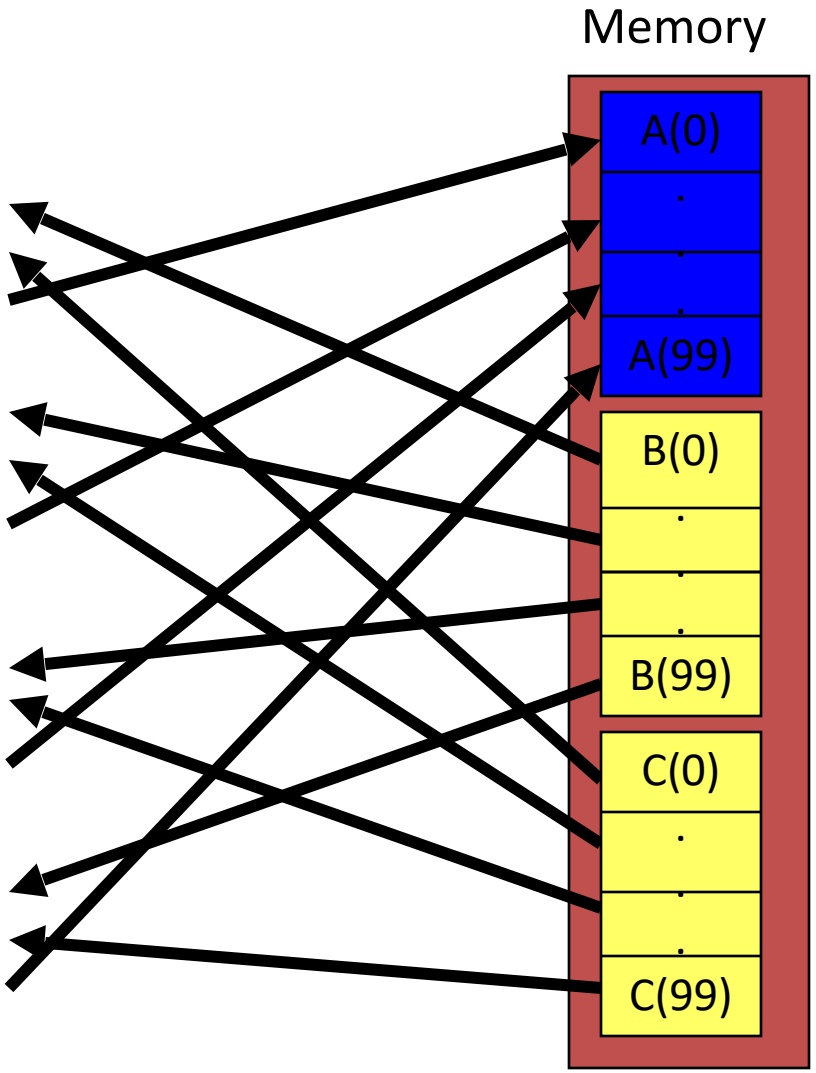
```
do i = 25, 49
    a(i) = b(i) + c(i)
end do
```

Thread 3

```
do i = 50, 74
    a(i) = b(i) + c(i)
end do
```

Thread 4

```
do i = 75, 99
    a(i) = b(i) + c(i)
end do
```



The Barrier Construct

- OpenMP `barrier` (implicit or explicit)
  - Threads wait until all threads of the current *Team* have reached the barrier

C/C++

```
#pragma omp barrier
```

- All worksharing constructs contain an implicit barrier at the end

The Single Construct

C/C++

```
#pragma omp single [clause]
... structured block ...
```

Fortran

```
!$omp single [clause]
... structured block ...
!$omp end single
```

- The `single` construct specifies that the enclosed structured block is executed by only on thread of the team.
  - It is up to the runtime which thread that is.

- Useful for:
  - I/O
  - Memory allocation and deallocation, etc. (in general: setup work)
  - Implementation of the single-creator parallel-executor pattern as we will see later…

The ~~Master~~ Construct (will be deprecated in OpenMP 6.0)

C/C++

```
#pragma omp master[clause]
... structured block ...
```

Fortran

```
!$omp master[clause]
... structured block ...
!$omp end master
```

- ~~The `master` construct specifies that the enclosed structured block is executed only by the master thread of a team.~~
  - Replacement: see the `masked` construct later

- Note: The masked construct is no worksharing construct and does not contain an implicit barrier at the end.

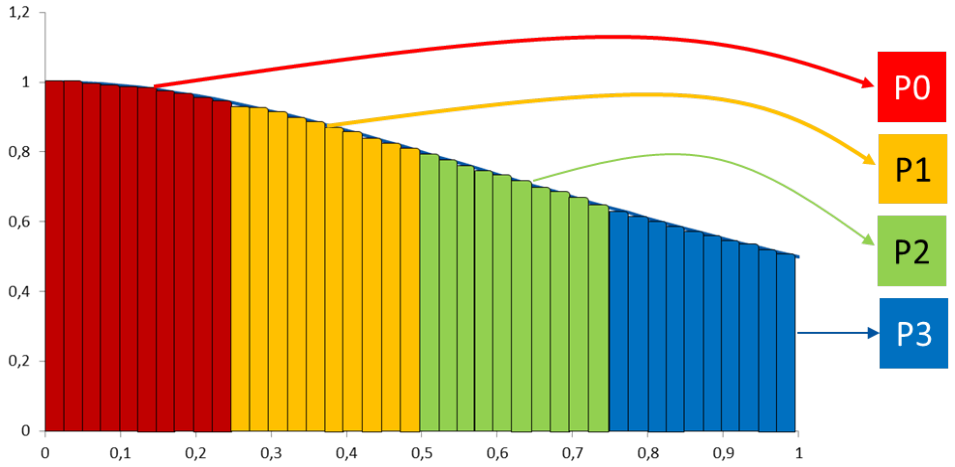
Vector Addition

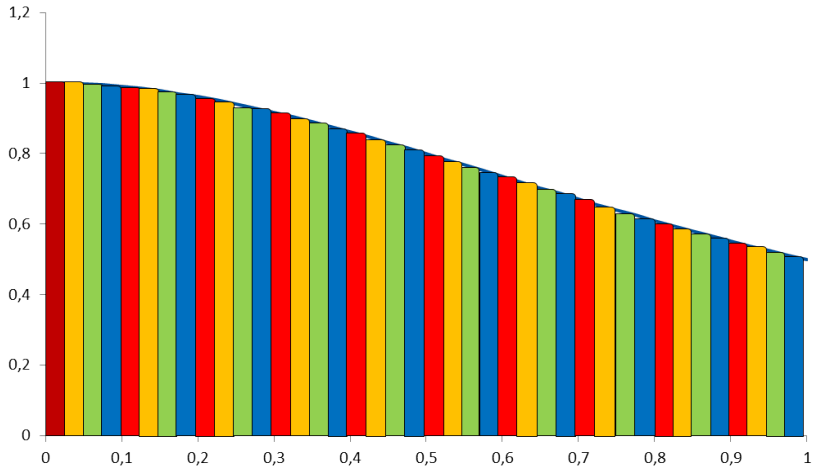
Influencing the For Loop Scheduling / 1

- *for*-construct: OpenMP allows to influence how the iterations are scheduled among the threads of the team, via the *schedule* clause:
  - `schedule(static [, chunk])`: Iteration space divided into blocks of chunk size, blocks are assigned to threads in a round-robin fashion. If chunk is not specified: #threads blocks.
  - `schedule(dynamic [, chunk])`: Iteration space divided into blocks of chunk (not specified: 1) size, blocks are scheduled to threads in the order in which threads finish previous blocks.
  - `schedule(guided [, chunk])`: Similar to dynamic, but block size starts with implementation-defined value, then is decreased exponentially down to chunk.
- Default is `schedule(static)`.

Influencing the For Loop Scheduling / 2

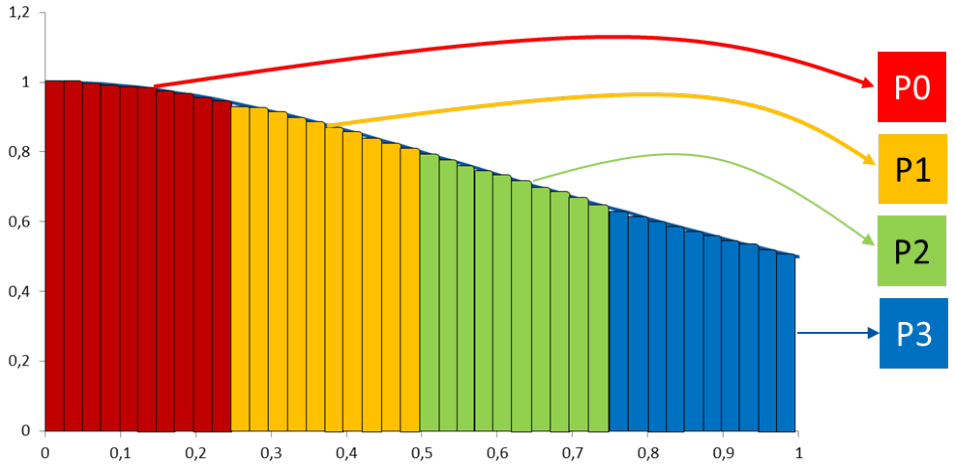
- **Static Schedule**
  - → `schedule(static [, chunk])`
  - → Decomposition depending on chunksize
  - → Equal parts of size ‘chunksize’ distributed in round-robin fashion

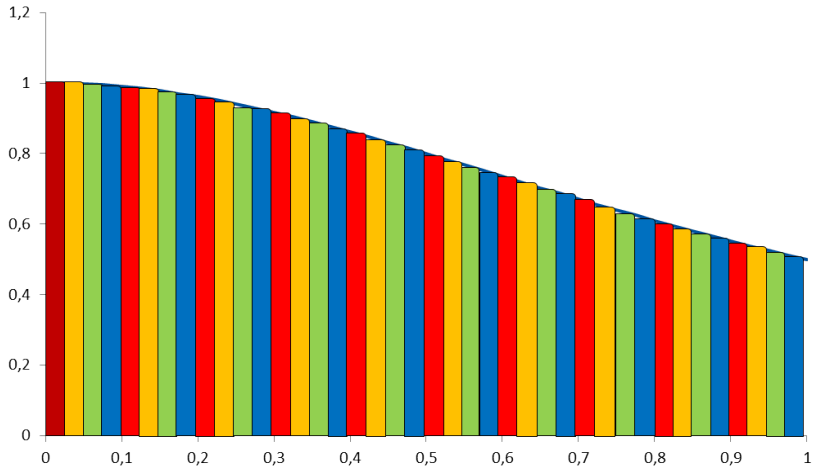




Influencing the For Loop Scheduling / 2

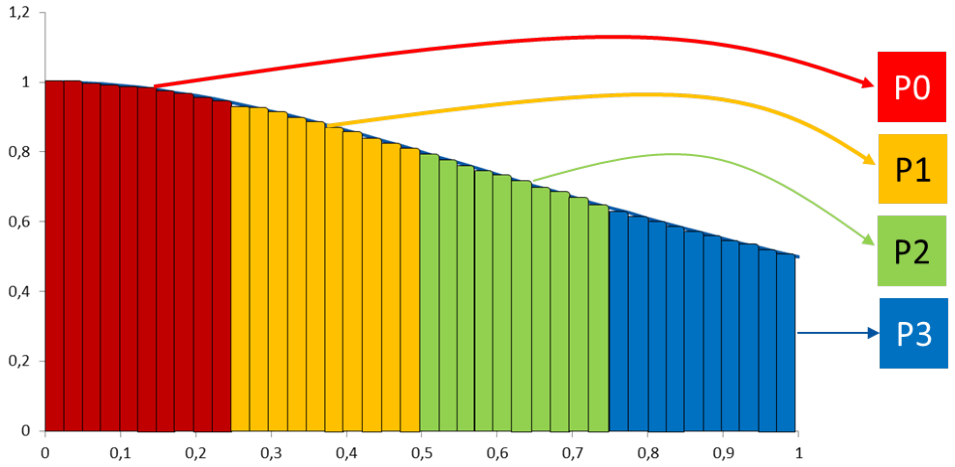
- **Static Schedule**
  - → `schedule(static [, chunk])`
  - → Decomposition depending on chunksize
  - → Equal parts of size ‘chunksize’ distributed in round-robin fashion
- **Pros?**

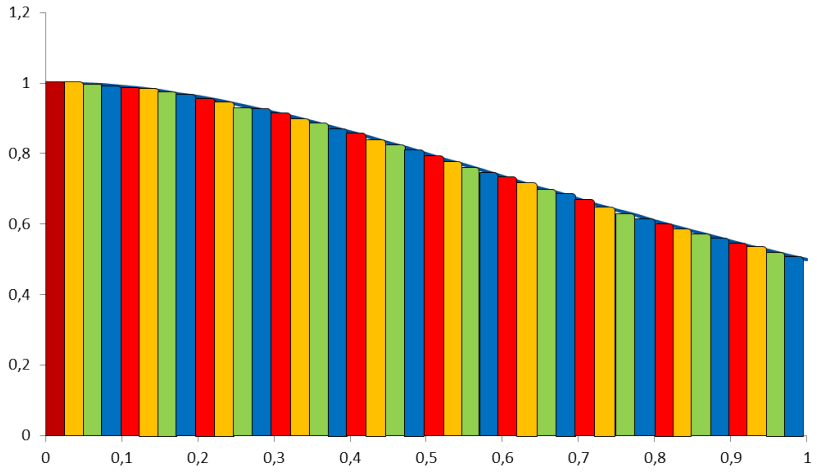




Influencing the For Loop Scheduling / 2

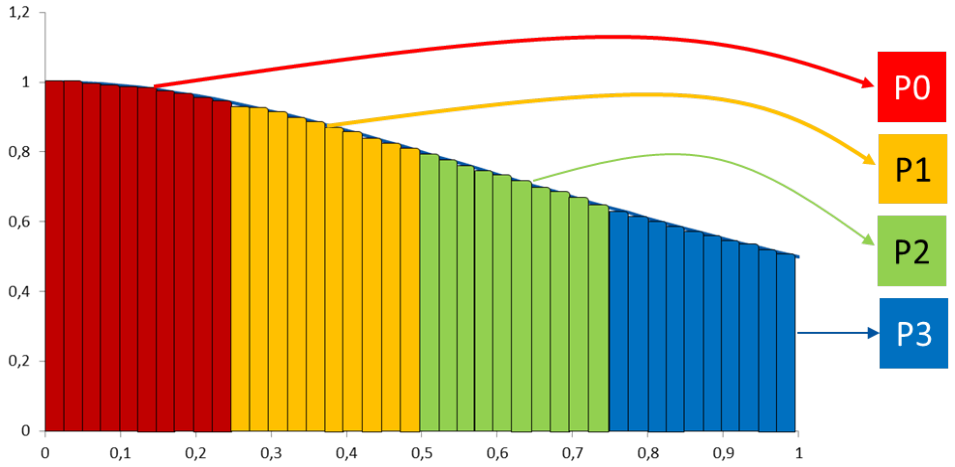
- **Static Schedule**
  - → `schedule(static [, chunk])`
  - → Decomposition depending on chunksize
  - → Equal parts of size ‘chunksize’ distributed in round-robin fashion
- **Pros?**
  - → No/low runtime overhead

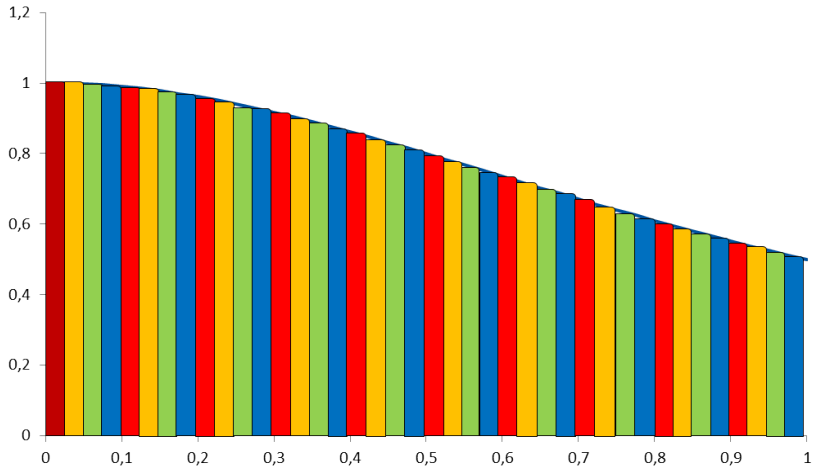




Influencing the For Loop Scheduling / 2

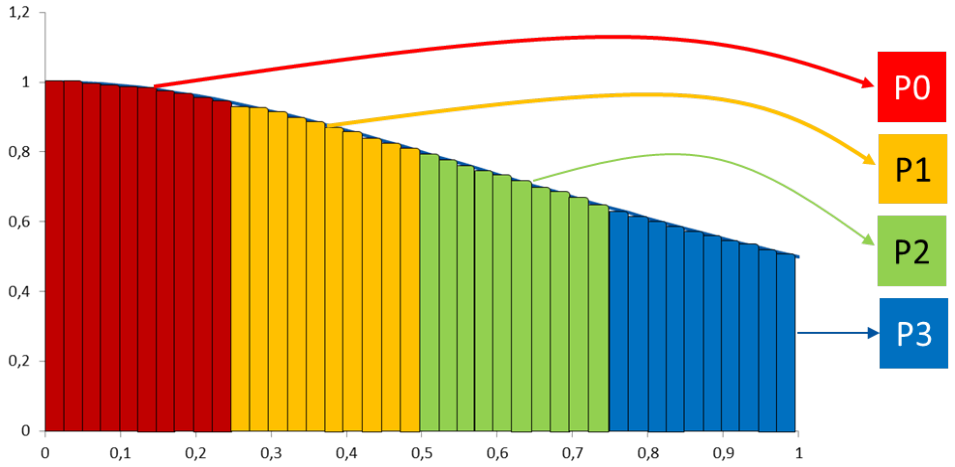
- **Static Schedule**
  - → `schedule(static [, chunk])`
  - → Decomposition depending on chunksize
  - → Equal parts of size ‘chunksize’ distributed in round-robin fashion
- **Pros?**
  - → No/low runtime overhead
- **Cons?**

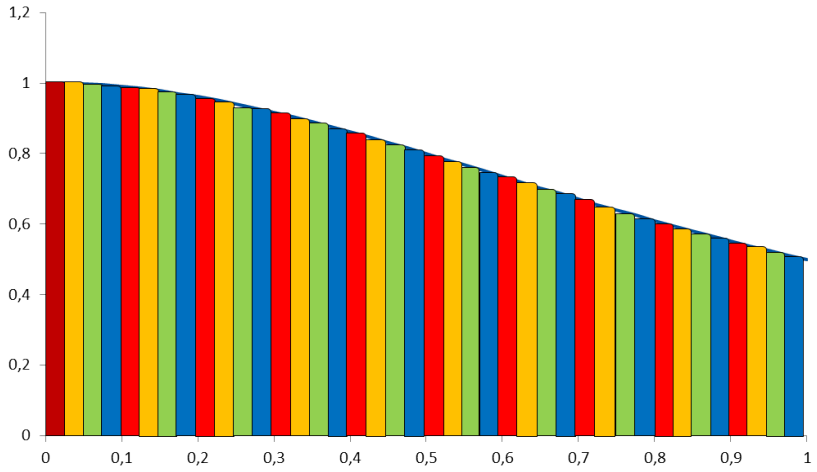




Influencing the For Loop Scheduling / 2

- **Static Schedule**
  - → `schedule(static [, chunk])`
  - → Decomposition depending on chunksize
  - → Equal parts of size ‘chunksize’ distributed in round-robin fashion
- **Pros?**
  - → No/low runtime overhead
- **Cons?**
  - → No dynamic workload balancing





Influencing the For Loop Scheduling / 3

- Dynamic schedule
  - `schedule(dynamic [, chunk])`
  - Iteration space divided into blocks of chunk size
  - Threads request a new block after finishing the previous one
  - Default chunk size is 1

Influencing the For Loop Scheduling / 3

- Dynamic schedule
  - `schedule(dynamic [, chunk])`
  - Iteration space divided into blocks of chunk size
  - Threads request a new block after finishing the previous one
  - Default chunk size is 1
- Pros ?

Influencing the For Loop Scheduling / 3

- Dynamic schedule
  - `schedule(dynamic [, chunk])`
  - Iteration space divided into blocks of chunk size
  - Threads request a new block after finishing the previous one
  - Default chunk size is 1
- Pros ?
  - Workload distribution

Influencing the For Loop Scheduling / 3

- Dynamic schedule
  - `schedule(dynamic [, chunk])`
  - Iteration space divided into blocks of chunk size
  - Threads request a new block after finishing the previous one
  - Default chunk size is 1
- Pros ?
  - Workload distribution
- Cons?

Influencing the For Loop Scheduling / 3

- Dynamic schedule
  - `schedule(dynamic [, chunk])`
  - Iteration space divided into blocks of chunk size
  - Threads request a new block after finishing the previous one
  - Default chunk size is 1
- Pros ?
  - Workload distribution
- Cons?
  - Runtime Overhead
  - Chunk size essential for performance
  - No NUMA optimizations possible

Synchronization Overview

- Can all loops be parallelized with `for`-constructs? No!
  - Simple test: If the results differ when the code is executed backwards, the loop iterations are not independent. BUT: This test alone is not sufficient:

C/C++

```
int i, int s = 0;

#pragma omp parallel for
for (i = 0; i < 100; i++)
{
   s = s + a[i];
}
```

- *Data Race*: If between two synchronization points at least one thread writes to a memory location from which at least one other thread reads, the result is not deterministic (race condition).

Synchronization: Critical Region

- A *Critical Region* is executed by all threads, but by only one thread simultaneously (*Mutual Exclusion*).

C/C++

```
#pragma omp critical (name)
{
   ... structured block ...
}
```

- Do you think this solution scales well?

C/C++

```
int i, s = 0;
#pragma omp parallel for
for (i = 0; i < 100; i++)
{
#pragma omp critical
   {  s = s + a[i];  }
}
```

Programming OpenMP

Scoping

Christian Terboven

Michael Klemm

Scoping Rules

- Managing the Data Environment is the challenge of OpenMP.

- *Scoping* in OpenMP: Dividing variables in *shared* and *private*:
  - *private*-list and *shared*-list on Parallel Region
  - *private*-list and *shared*-list on Worksharing constructs
  - General default is *shared* for Parallel Region, *firstprivate* for Tasks.
  - Loop control variables on *for*-constructs are *private*
  - Non-static variables local to Parallel Regions are *private*
  - *private*: A new uninitialized instance is created for the task or each thread executing the construct
    - *firstprivate*: Initialization with the value before encountering the construct
    - *lastprivate*: Value of last loop iteration is written back to the initial thread
  - Static variables are *shared*

Scoping Rules

- Managing the Data Environment is the challenge of OpenMP.

- *Scoping* in OpenMP: Dividing variables in *shared* and *private*:
  - *private*-list and *shared*-list on Parallel Region
  - *private*-list and *shared*-list on Worksharing constructs
  - General default is *shared* for Parallel Region, *firstprivate* for Tasks.
  - Loop control variables on *for*-constructs are *private*
  - Non-static variables local to Parallel Regions are *private*
  - *private*: A new uninitialized instance is created for the task or each thread executing the construct
    - *firstprivate*: Initialization with the value before encountering the construct
    - *lastprivate*: Value of last loop iteration is written back to the initial thread
  - Static variables are *shared*

Tasks are introduced later

Privatization of Global/Static Variables

- Global / static variables can be privatized with the *threadprivate* directive
  - One instance is created for each thread
    - Before the first parallel region is encountered
    - Instance exists until the program ends
    - Does not work (well) with nested Parallel Region
  - Based on thread-local storage (TLS)
    - TlsAlloc (Win32-Threads), pthread_key_create (Posix-Threads), keyword `__thread` (GNU extension)

C/C++

```
static int i;
#pragma omp threadprivate(i)
```

Fortran

```
SAVE INTEGER :: i
!$omp threadprivate(i)
```

Privatization of Global/Static Variables

- Global / static variables can be privatized with the *threadprivate* directive
  - One instance is created for each thread
    - Before the first parallel region is encountered
    - Instance exists until the program ends
    - Does not work (well) with nested Parallel Region
  - Based on thread-local storage (TLS)
    - TlsAlloc (Win32-Threads), pthread_key_create (Posix-Threads), keyword `__thread` (GNU extension)

Really: try to avoid the use of threadprivate and static variables!

C/C++

```
static int i;
#pragma omp threadprivate(i)
```

Fortran

```
SAVE INTEGER :: i
!$omp threadprivate(i)
```

Back to our example

C/C++

```
int i, s = 0;
#pragma omp parallel for
for (i = 0; i < 100; i++)
{
#pragma omp critical
   {  s = s + a[i];  }
}
```

It‘s your turn: Make It Scale!

```
#pragma omp parallel
{

#pragma omp for
   for (i = 0; i < 99; i++)
   {

         s  = s   + a[i];

   }

} // end parallel
```

```
do i = 0, 99
    s = s + a(i)
end do
```

→

```
do i = 0, 24
    s = s + a(i)
end do
```

```
do i = 25, 49
    s = s + a(i)
end do
```

```
do i = 50, 74
    s = s + a(i)
end do
```

```
do i = 75, 99
    s = s + a(i)
end do
```

(done)

```
#pragma omp parallel
{
    double ps = 0.0;    // private variable
#pragma omp for
   for (i = 0; i < 99; i++)
   {
         ps = ps + a[i];
   }
#pragma omp critical
{
    s += ps;
}
} // end parallel
```

```
do i = 0, 99
    s = s + a(i)
end do
```

→

```
do i = 0, 24
    s1 = s1 + a(i)
end do
s = s + s1
```

```
do i = 25, 49
    s2 = s2 + a(i)
end do
s = s + s2
```

```
do i = 50, 74
    s3 = s3 + a(i)
end do
s = s + s3
```

```
do i = 75, 99
    s4 = s4 + a(i)
end do
s = s + s4
```

The Reduction Clause

- In a *reduction*-operation the operator is applied to all variables in the list. The variables have to be *shared*.
  - `reduction(operator:list)`
  - The result is provided in the associated reduction variable

C/C++

```
int i, s = 0;

#pragma omp parallel for reduction(+:s)
for(i = 0; i < 99; i++)
{
   s = s + a[i];
}
```

  - Possible reduction operators with initialization value:
    `+ (0), * (1), - (0), & (~0), | (0), && (1), || (0), ^ (0), min (largest number), max (least number)`
  - Remark: OpenMP also supports user-defined reductions (not covered here)

PI

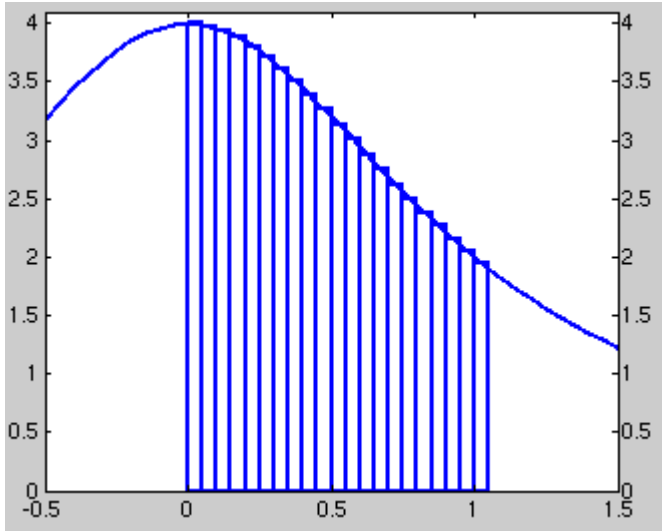
Example: Pi (1/2)

```
double f(double x)
{
   return (4.0 / (1.0 + x*x));
}

double CalcPi (int n)
{
   const double fH   = 1.0 / (double) n;
   double fSum = 0.0;
   double fX;
   int i;

#pragma omp parallel for
   for (i = 0; i < n; i++)
   {
      fX = fH * ((double)i + 0.5);
      fSum += f(fX);
   }
   return fH * fSum;
}
```

$$\pi = \int_0^1 \frac{4}{1+x^2}$$



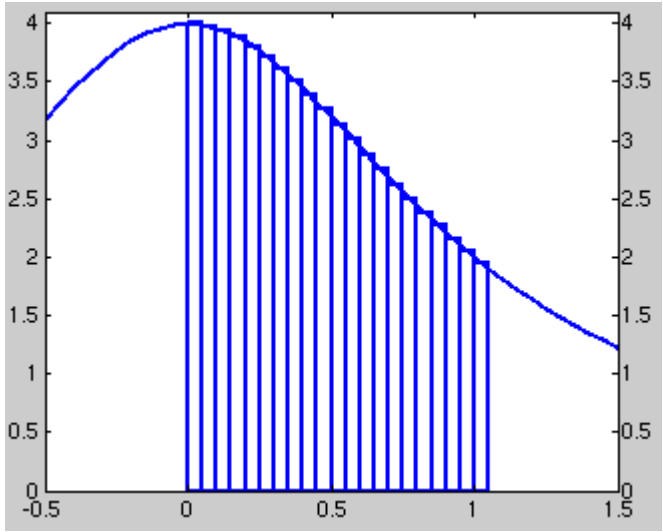
Example: Pi (2/2)

```
double f(double x)
{
   return (4.0 / (1.0 + x*x));
}

double CalcPi (int n)
{
   const double fH   = 1.0 / (double) n;
   double fSum = 0.0;
   double fX;
   int i;

#pragma omp parallel for private(fX,i) reduction(+:fSum)
   for (i = 0; i < n; i++)
   {
      fX = fH * ((double)i + 0.5);
      fSum += f(fX);
   }
   return fH * fSum;
}
```

$$\pi = \int_0^1 \frac{4}{1+x^2}$$



Programming OpenMP

OpenMP Tasking Introduction

Christian Terboven

Michael Klemm

Tasking Execution Model

- Supports unstructured parallelism
  → unbounded loops

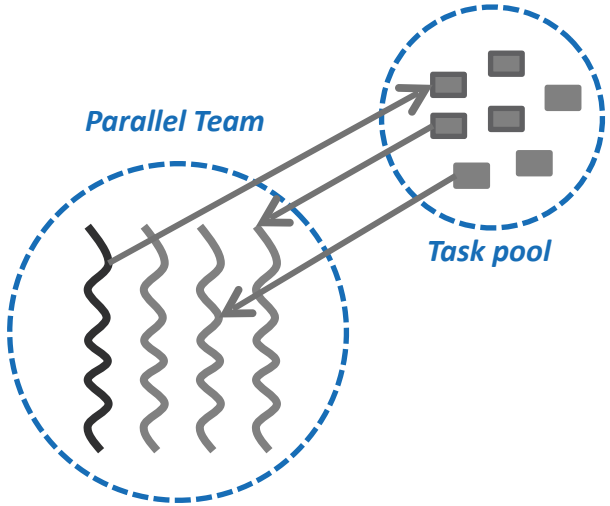
```
while ( <expr> ) {
    ...
}
```

→ recursive functions

```
void myfunc( <args> )
{
    ...; myfunc( <newargs> ); ...;
}
```

- Example (unstructured parallelism)

```
#pragma omp parallel
#pragma omp masked
while (elem != NULL) {
    #pragma omp task
        compute(elem);
    elem = elem->next;
}
```



- Several scenarios are possible:
  → single creator, multiple creators, nested tasks (tasks & WS)
- All threads in the team are candidates to execute tasks

What is a Task in OpenMP?

- Tasks are work units whose execution
  - → may be deferred or…
  - → … can be executed immediately
- Tasks are composed of
  - → **code** to execute, a **data** environment (initialized at creation time), internal **control** variables (ICVs)
- Tasks are created…

  … when reaching a parallel region → implicit tasks are created (per thread)

  … when encountering a task construct → explicit task is created

  … when encountering a taskloop construct → explicit tasks per chunk are created

  … when encountering a target construct → target task is created

OpenMP Tasking Idiom

- OpenMP programmers need a specific idiom to kick off task-parallel execution: `parallel masked`
  - → OpenMP version 5.0 introduced the `parallel master` construct
  - → With OpenMP version 5.1 this becomes `parallel masked`

```
 1  int main(int argc, char* argv[])
 2  {
 3      [...]
 4      #pragma omp parallel
 5      {
 6         #pragma omp masked
 7          {
 9
start_task_parallel_execution();
 9          }
10      }
11.     [...]
12.}
```

```
 1  int main(int argc, char* argv[])
 2  {
 3      [...]
 4      #pragma omp parallel
 5      {
 6         #pragma omp single
 7          {
 9
start_task_parallel_execution();
 9          }
10      }
11.     [...]
12.}
```

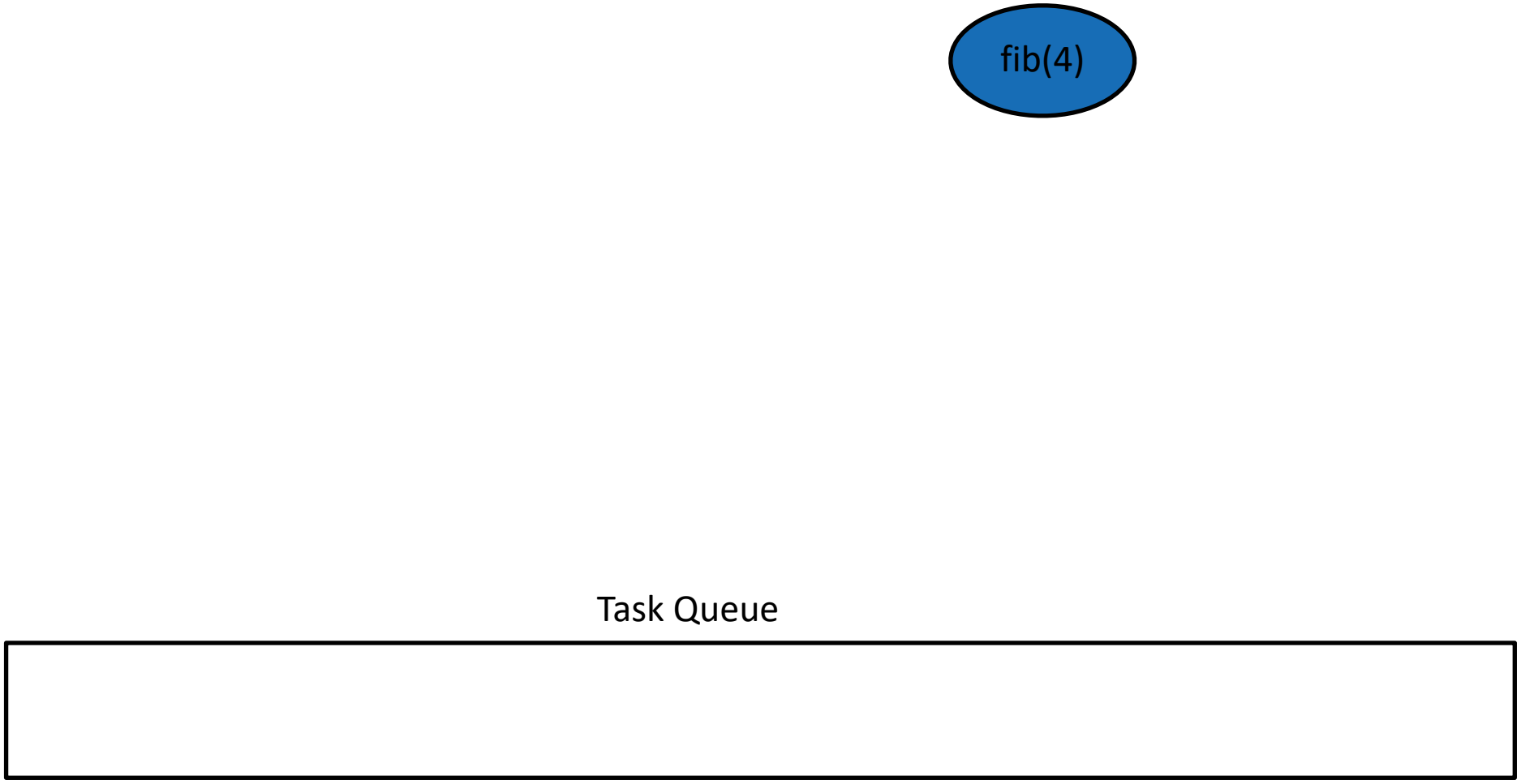
Fibonacci Numbers (in a Stupid Way ☺)

```
 1  int main(int argc,
 2           char* argv[])
 3  {
 4      [...]
 5      #pragma omp parallel
 6      {
 7         #pragma omp masked
 8         {
 9             fib(input);
10         }
11      }
12      [...]
13.}
```

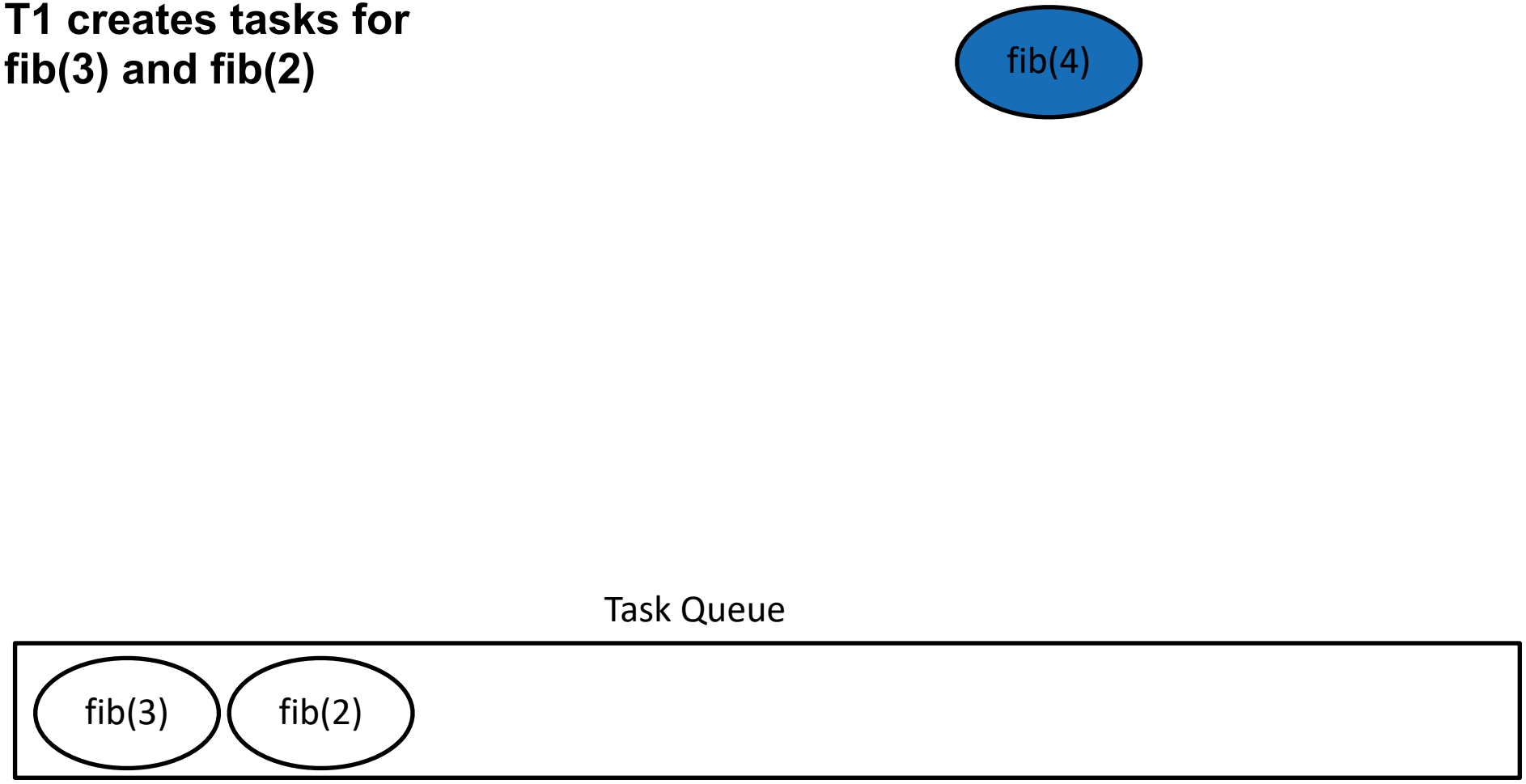
```
14  int fib(int n)   {
15      if (n < 2) return n;
16      int x, y;
17      #pragma omp task shared(x)
18      {
19          x = fib(n - 1);
20      }
21      #pragma omp task shared(y)
22      {
23          y = fib(n - 2);
24      }
25      #pragma omp taskwait
26          return x+y;
27.}
```

- Only one thread enters `fib()` from `main()`.
- That thread creates the two initial work tasks and starts the parallel recursion.
- The `taskwait` construct is required to wait for the result for `x` and `y` before the task can sum up.

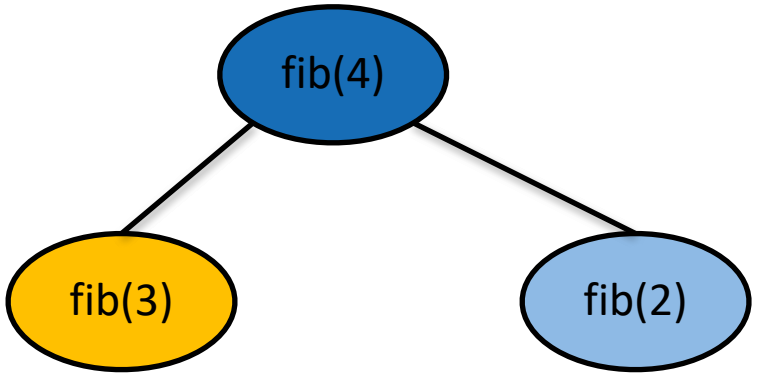
- **T1 enters fib(4)**



- **T1 enters fib(4)**
- **T1 creates tasks for fib(3) and fib(2)**

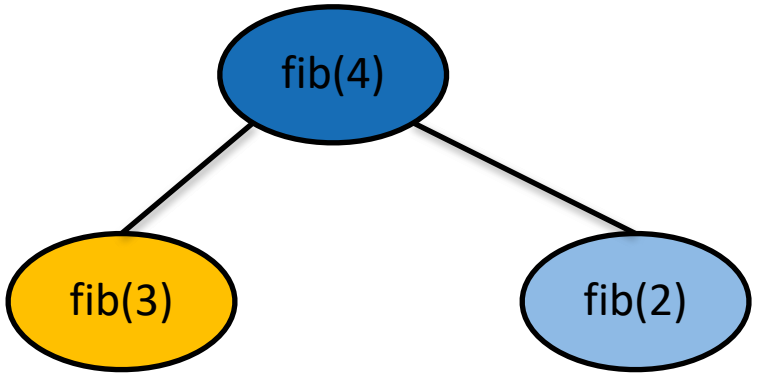


- **T1 enters fib(4)**
- **T1 creates tasks for fib(3) and fib(2)**
- **T1 and T2 execute tasks from the queue**

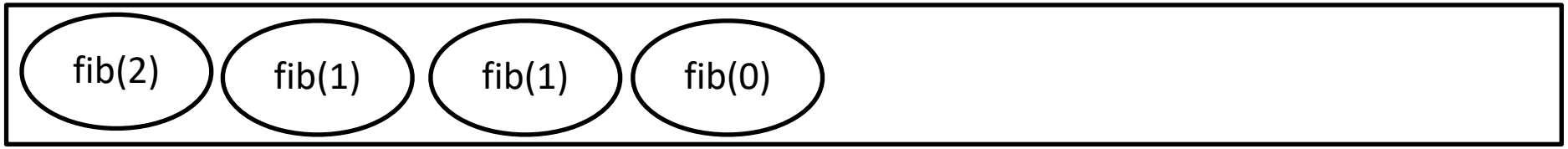


Task Queue

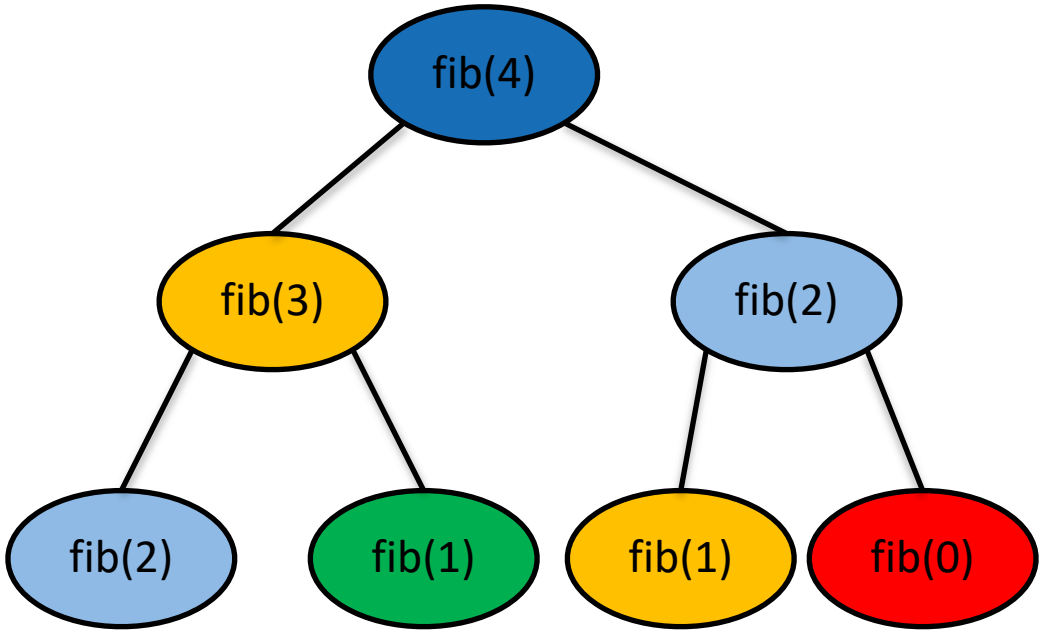
- T1 enters fib(4)
- T1 creates tasks for fib(3) and fib(2)
- T1 and T2 execute tasks from the queue
- T1 and T2 create 4 new tasks



Task Queue

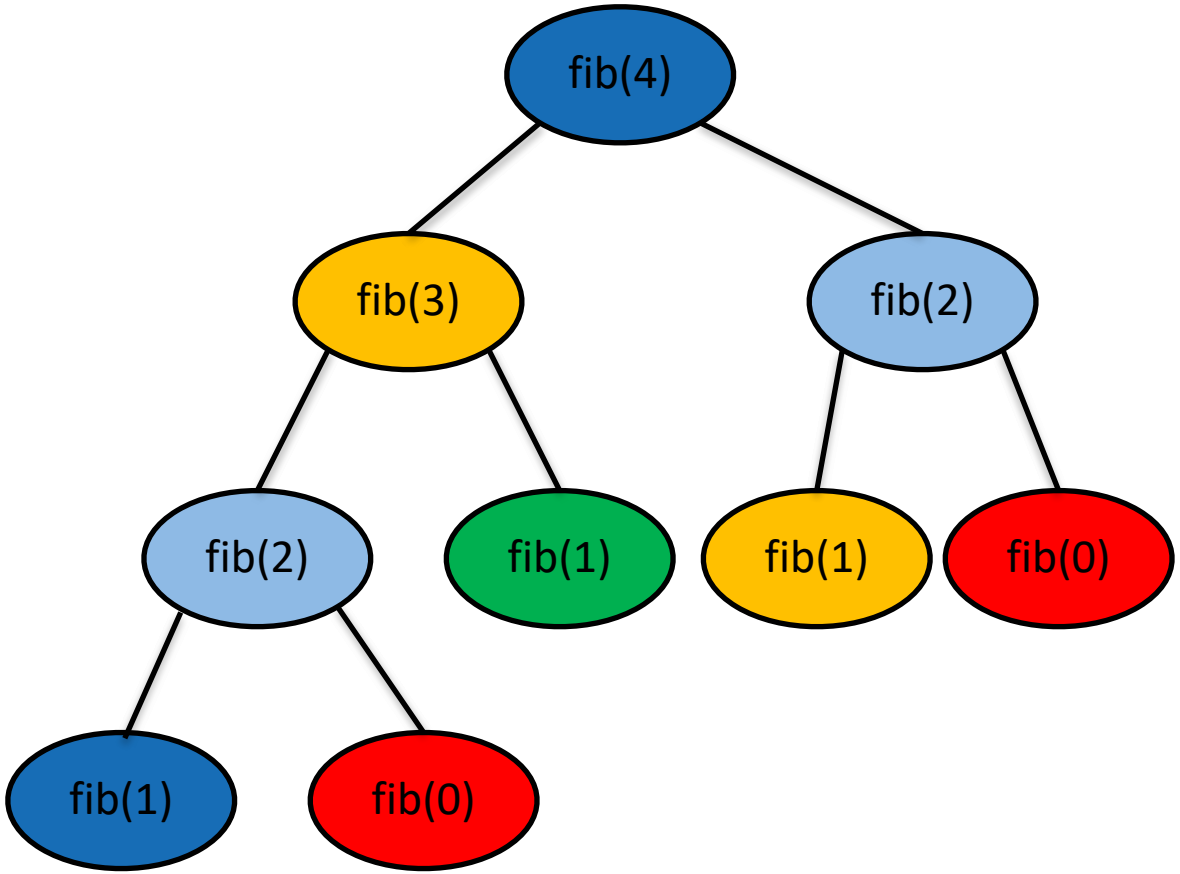


- T1 enters fib(4)
- T1 creates tasks for fib(3) and fib(2)
- T1 and T2 execute tasks from the queue
- T1 and T2 create 4 new tasks
- T1 - T4 execute tasks



Task Queue

- T1 enters fib(4)
- T1 creates tasks for fib(3) and fib(2)
- T1 and T2 execute tasks from the queue
- T1 and T2 create 4 new tasks
- T1 - T4 execute tasks
- …



Programming OpenMP

Hands-on Exercises

Christian Terboven

Michael Klemm

RWTH AACHEN UNIVERSITY

Exercises

- We have implemented a series of small hands-on examples that you can use and play with.
  - → Download: https://github.com/NERSC/openmp-series-2024
  - → Build: `make`

- Each hands-on exercise has a folder “solution”
  - → It shows the OpenMP solution that we have added
  - → You can use it to cheat ☺, or to check if you came up with the same solution

Exercises: Overview

Exercise no.	Exercise name	OpenMP Topic	Day / Order (proposal)
1	Hello World	Getting started	Start with this (if OpenMP is new for you)
2	Pi	Worksharing, Scoping	First day
3	Jacobi	Worksharing, Scoping	First day
4	Work-Distribution	Worksharing	First day
5	Min/Max	Worksharing, Reduction	First day

to be continued …